Hashmap (hashtable)

A hashmap (also known as a Hashtable) is a data structure that stores key-value pairs.

It allows fast retrieval of values based on the keys by using a process called 'Hashing'.

Each key is transformed into a unique index through a hash function, which can be used to access the value (in constant time).

→ Real world example

1. Dictionary
2. Phone contacts
3. Student ID

→ Practical examples

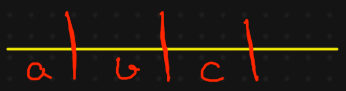
1. DNS lookup
2. No sql DBs
3. Caching in Web Browser
4. Symbol table in Compiler

Why Hashing ?

→ hashmaps are pretty awesome.

Which is the highest occuring alphabet ?

1. Sort the string ⟶ O(nlogn)



2. count = [0] * 26

count [0]

count ['a'] ✗

a → 1
b → 2
⋮
z → 26
finite O(n)

list can just have integer as keys.

Q: highest occuring word in the sentence?

words are infinite

count [`in'] ✗

a → 1
aa → 2
ab → 3

In arr/list only integers are allowed as keys, but we want to decide the key

count [`key'] = value

(control both. (any format)

Operations

→ add / update (key, value)
→ search/get (key)
→ delete (key)

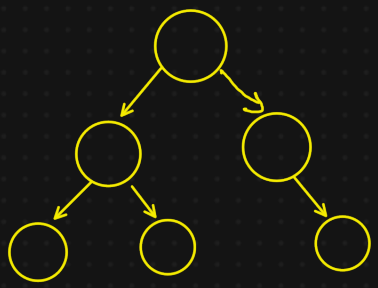
1. Linked List

Key, value
next

1. Insert/update
2. Search
3. Delete

Each operation is O(n)

2. BST



balanced

O(log n)

insert
delete
search

3. Hashmaps

O(1)

Inbuilt hashmap in Python

```
dict = { }
```

we can store things in key-value pair

Please make sure that you try out the questions given

Implementing our own hashmaps : Hashing

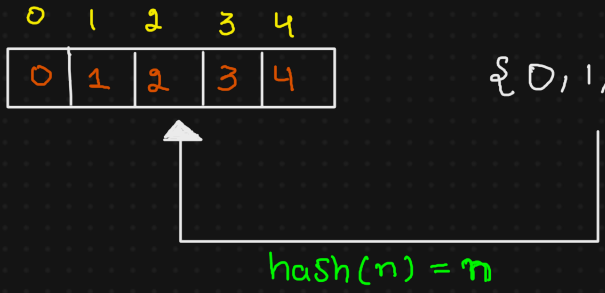
list

access via indexing is O(1) operation

a[5]

independent of len

O(1)



how to store for O(1) retrieval ?

→ store each element at index equal to element

Bucket array

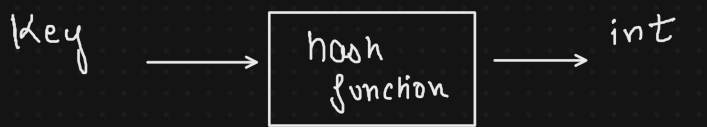
{0, 1, 2, 3, 4, 255873}

A lot of memory wastage if we take a big array/list

inefficient

waste of memory

Key can be anything & our aim should be to somehow change the given key into an integer,



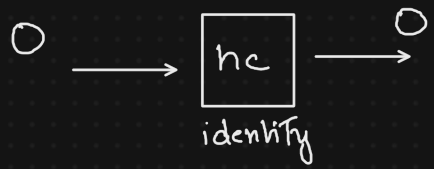
Hash function

1. Hash code → Key converted to integer
2. Compress → Key compressed to have an index present in bucket array.

%

% n → 0 - n-1

We can have any hash code which converts key to integer



'abc' → we can use the ascii values 97+98+99

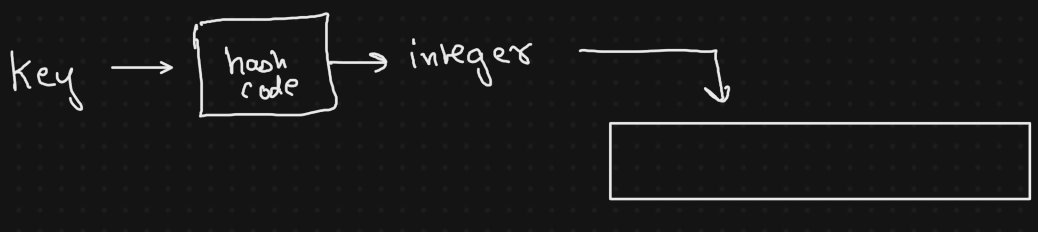
Uniform distribution is there not concentrated

prime numbers have good distribution

$321 \rightarrow 3 * \underline{10}^2 + 2 * \underline{10}^1 + 1 * \underline{10}^0$ → spread across
P P P

abc → 'a' * p^3 + 'b' * p^2 ... → very large values

What if a node has to be stored as key?



% 5

← 10 numbers

→ 100025 = 0

→ 2002075 = 0

% 5

Above is known as Collision

Collision Handling

50	43	47	60	80	%5
0	3	2	0	0	

1. Open Addressing (Closed hashing)

no. of elements ≤ arrayList size

If inital place is empty save it
else find an alternative.

59	43	47	69	80	%5
0	3	2	0	0	
4			4		

$$g(i) = [h(i) + K(i')] \% size$$

- $g(i)$ → final index
- $K(i')$ → f^n = diff $f(i)$
- i' = tries

69 %5 = 4

(4+1) %5

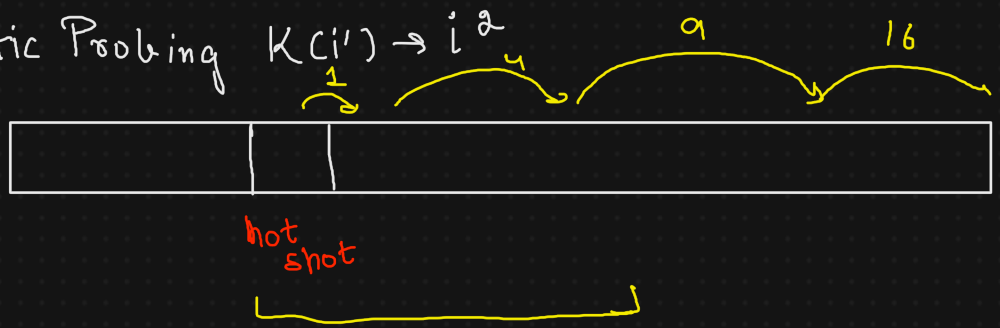
5 %5 → 0

1. Linear Probing $K(i') \rightarrow i$

$$g(i) = h(i) + 0$$
$$= h(i) \% size$$

for each increasing try we increase hash f^n linearly.

2. Quadratic Probing $K(i') \rightarrow i^2$



1 → 1
2 → 4
3 → 9

iii) Double hashing $f(i) \rightarrow i * h'(i)$

$$g(i) = [h(i) + f(i)] \% \text{ size}$$

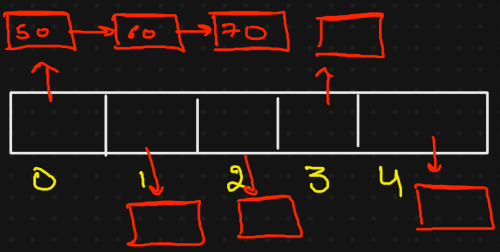
$f(i)$ ⇓ another hash function

2. Closed addressing (Chaining)

50, 42, 49, 60, 70

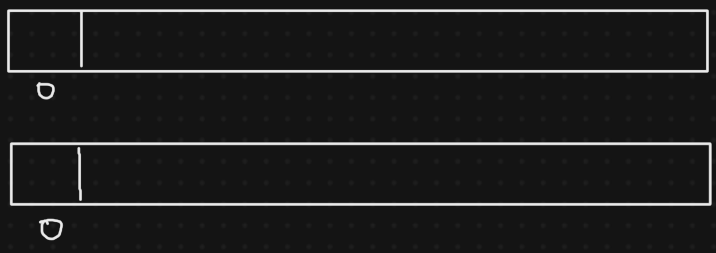
%5

0 2 4 0 0



we have each position of bucket array as head of linked list

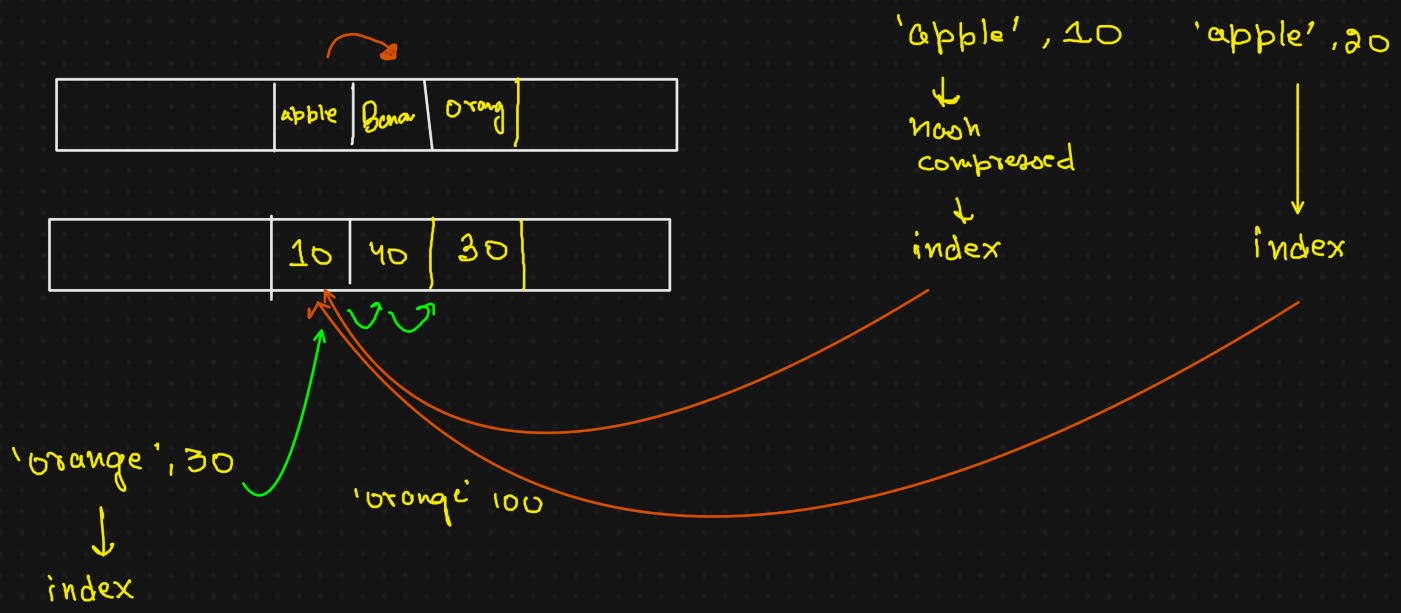
Hashmap Implementation - Open Addressing



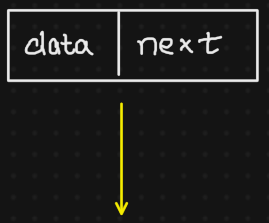
Capacity
size

'apple' , 10
Key value

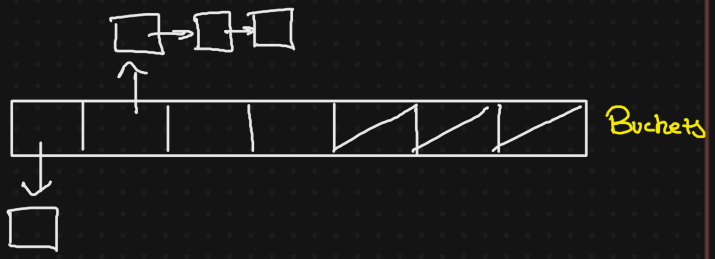
Insert



Hashmap Implementation :- Chaining



we will also have a Key and a value

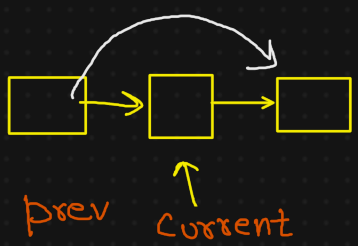


Linked List operation

delete

(i) if node is head

(ii)



Buckets list

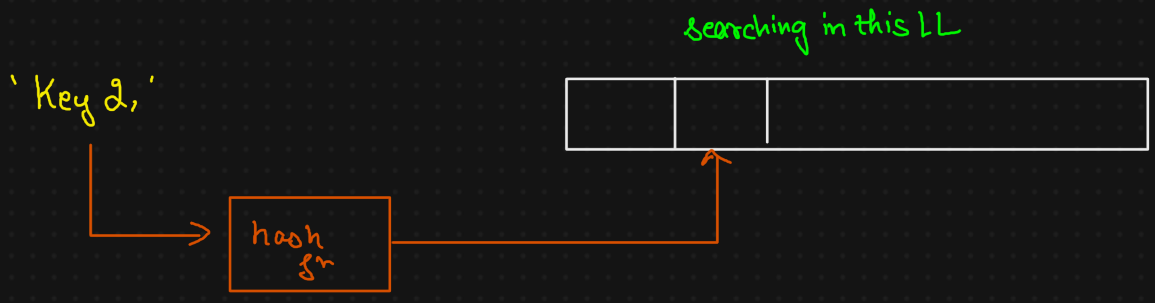
[None] * Capacity

Should be a seperate head of our new linked list class

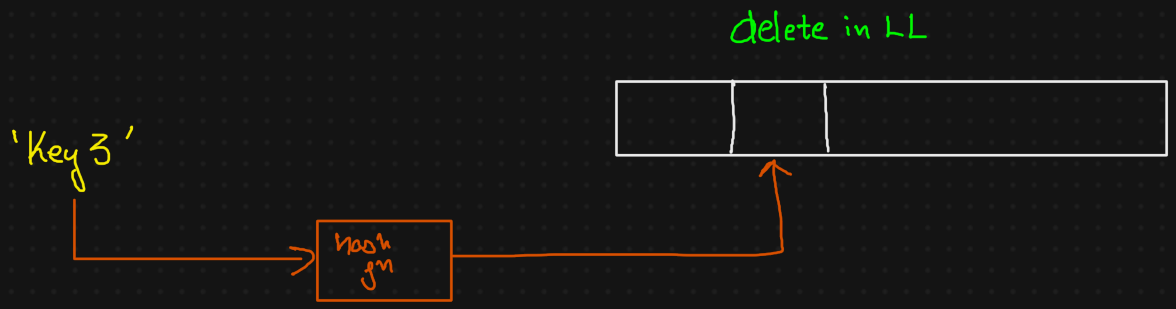
Insert

'Key1', 100 → hash fn → bucket index →

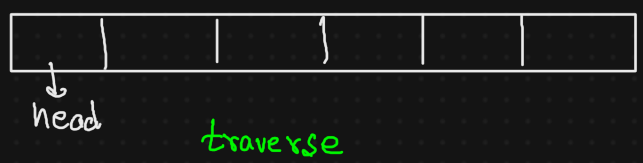
Search



Delete



Print

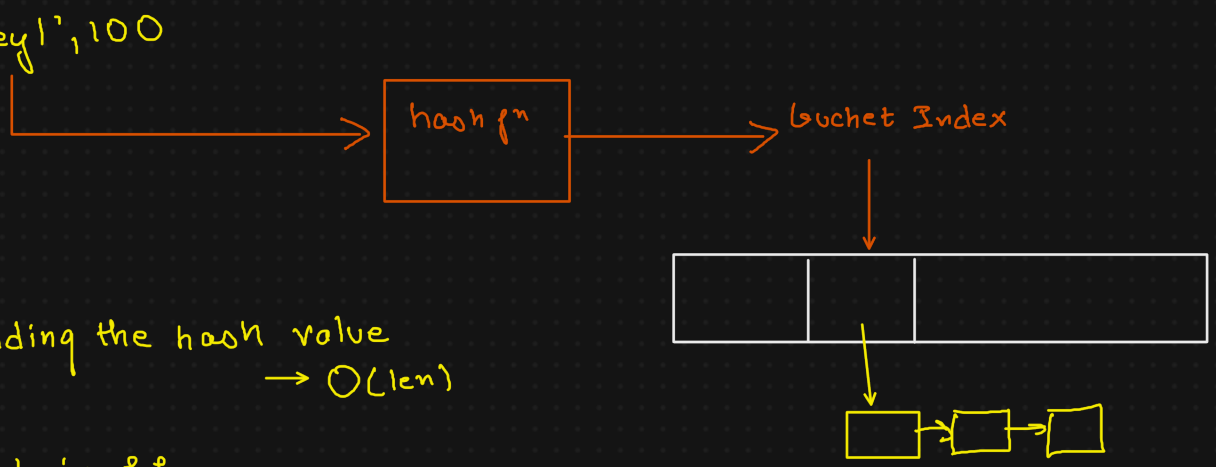


Complexity Analysis of our Hashmap

1. Insert

$n \rightarrow$ no. of entries

$b \rightarrow$ bucket size



1. finding the hash value $\rightarrow O(len)$
2. Insert in LL

$O(n)$

1. In worst case all can come to a same bucket index, search will become $O(n)$
2. $n >>>$ len, so hashing can be $O(1)$ is constant

normally due to good hash codes. we don't get this bad of a scenario and get a uniform distribution

$O(\frac{n}{b})$

$O(1)$

$\frac{n}{b}$ = load factor ~ 0.75

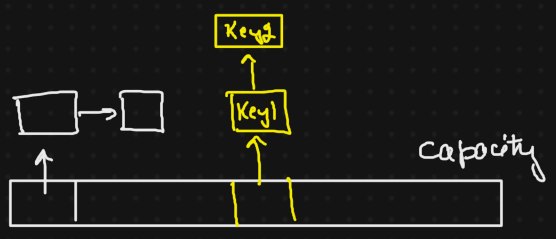
we have to increase b to keep load factor in check

old_b = 10

'Key1' → 105 % 10 = 5

'Key' → 115 % 10 = 5

$\frac{\text{size}}{\text{capacity}} > 0.75$ old

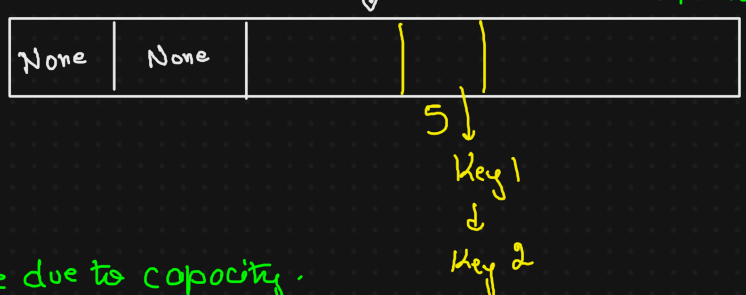


new_b = 20

'Key1' → 105 % 20 = 5

'Key2' → 115 % 20 = 15

new bucket



The final bucket index will change due to capacity.

Rehashing → to keep load factor ($\frac{n}{b}$) all entry are hashed to a new bucket array.

So, time complexity remain $O(1) \sim O(n/b)$

Rehashing

1. Save oldbuckets
2. double capacity
3. new-buckets
4. self.size = 0
5. for every entry
   just insert